PROG 3

Intro

C++ offers many language features, such as

- Procedural programming
- Object-oriented programming
- Generic meta programming
- Functional programming

Large code bases can be handled, C++ allows easy access to C APIs, allows low level optimizations, and is a very powerful language.

Hello World

Standard libary elements are in the **namespace** "std" and can be accessed with the **#include** keyword.
C++ compilers generate platform dependent binaries, f.e. Java is platform independent. C++ programs need to be compiled for each platform.

How to edit & compile

Edit a C++ file wit `gedit helloworld.cpp` and compile it with `g++ -c helloworld.cpp` .
Link the file and build an executable with `g++ helloworld.o -o helloworld.exe` . –> `./helloworld.exe`

Declarations & Definitions

Declarations introduce the existence of structures, variables, functions, etc. –> declare a function:

```
int add(int, int);
```

Definitions are declarations, which contain all information about the declared thing. –> define a function:

```
int add(int a, int b) {
    return a + b;
}
```

ODR (One Definition Rule)

- Only one single definition of a function, variable, class, etc. is allowed.
- Every used thing must be defined somewhere.

Redeclaration of a function is allowed, if the definition is the same.

Modularization

Header files contain declarations, which can be included in other files.

```
// helloworld.h
#ifndef HELLOWORLD_H
#define HELLOWORLD_H
    int add(int, int);
#endif
```

```
// helloworld.cpp
#include "helloworld.h"
    int add(int a, int b) {
        return a + b;
    }
```

Libaries are collections of header files, which can be included in other files. They can be either static (*.a*/.LIB) or dynamic (*.so*/.DLL).

Namespaces

Namespaces are used to avoid name collisions.

```
namespace mynamespace {
    int add(int a, int b) {
        return a + b;
    }
}

int main() {
    int a = 1;
    int b = 2;
    int c = mynamespace::add(a, b);
    return 0;
}
```

Makefiles

Makefiles are used to automate the build process.

CMake is a Makefile generator, which can be used to generate Makefiles for different platforms.

First Steps

Functions

- Functions can be defined for different types. –> **overloading**
- Function calls with ambiguous types are not allowed. –> **overloading resolution**

Variables, Narrowing

- Variables are defined prior usage.
- Initialization `a=2` is deprecated, use `a{2}` or `a={2}` instead.
- Narrowing: losing information during type conversion. –> `int a = 2.5;`
- Array variables are defined: `TYPE arr[NUM]`.
- C++11 defined std::array `std::array<TYPE, NUM> arr`.
- Array sizes must be known at compile time.

Constants

- Constants are defined with `const`.
- const variabes protect variables from modification.
- constexpr variables protect variables from modification and allow compile time evaluation.

Refrences & Pointers

Pointers Features:

- Pointer = address (where) + optional: type (what)
- Nullpointer = `nullptr`
- Pointer arithmetic: address modifications

Use cases:

- Data structures –> Lists
- Data referencing (passing pointers instead of values)
- Dynamic memory management

Pointer declaration `TYPE* name {...};`
Addresses of variables can be accessed with `&name`
Pointer arithmetic is possible –> `&c2-&c1`
To access the data to which a pointer is pointing use the dereference operator `*` –> `*name=2;`

```
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int main() {
    int x{2},y{3};
    int *xp = &x;
    swap(xp, &y);
    swap(&x, &y);
}
```

References

- Reference variable declaration: `TYPE& name{...};` - no reassignment possible
- References are aliases for variables
- Accessing a reference is the same as accessing the original value
- References can't be null

C-Strings

- C-Strings are arrays of characters,
- `const TYPE* ptr`is a pointer to a const TYPE
- `TYPE* const ptr`is a const pointer to a TYPE

Different function parameters **Pass by value** `func (TYPE value)` –> copy value (input, small TYPES)
Pass by reference:
`func (TYPE &value)` –> reference to original value (input, output) `func (const TYPE &value)` –> reference to original value (input, large TYPES)
Pass by pointer:
`func (TYPE *value)` –> reference to original value (input, output)
`func (const TYPE *value)` –> reference to original value (input)

Dynamic Memory Management

Allocation `TYPE* ptr = new TYPE{init};`
Deallocation `delete ptr;`
Allocate N data element `TYPE* ptr = new TYPE[N];`
Access `ptr[i]`
Deallocate `delete[] ptr;` **Dangling pointer** is a pointer, which points to a deallocated memory location.

I/O

Open a file `std::ifstream` **Read from a file** `std::getline(std::cin, line);` **Write to a file** `std::cout << "Hello World" << std::endl;`